TRAINING THE ORCHESTRATOR: VERIFIER-FILTERED SUPERVISION FOR AGENT SELECTION IN LLM-DRIVEN PDDL PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

People state planning goals in natural language, while classical planners require formal PDDL. Recent agentic frameworks bridge this gap with a verifier-checked refinement loop in which an orchestrator picks, at every step, which repair agent (an LLM-prompted or deterministic PDDL and plan-repair routine) to run next; that orchestrator is itself a prompted frontier LLM. We present HALO (Hybrid Agent-Learned Orchestrator), which replaces it with a QLoRA-tuned 8B model (Llama-3-8B) trained on the agent choices of a prompted teacher, keeping only trajectories whose final plan an external validator accepts. This is outcome-filtered behavioural cloning: the validator certifies where a trajectory ended, not each step, yet it is enough to make imitation work, and removing the filter collapses performance. HALO places the policy behind three hardcoded rules for trivially decidable steps and selects among an expanded pool of 21 agents; the rules are essential, as the policy alone falls well short. Across 11 PDDL domains from PlanBench, Natural Plan, and classical planning, and measuring success as validator acceptance of the generated PDDL and plan, HALO matches the GPT-5-mini-orchestrated framework on PlanBench and classical planning, exceeds it by 37 points on Natural Plan, and stays within three points of Gemini-3-Flash, while cutting orchestration cost roughly 45× and 15× respectively and LLM calls per episode by 40–54%.

1 INTRODUCTION

Automated planning is a cornerstone of artificial intelligence: from logistics and scheduling to robotics, intelligent systems must turn high-level goals into action sequences that achieve them reliably. Classical planning, formalised in the Planning Domain Definition Language (PDDL) (McDermott et al., 1998; Ghallab et al., 2004), provides strong guarantees (soundness, completeness, often cost optimality) once a task is expressed in its formal vocabulary of typed objects, predicates, actions, and goals. The cost of those guarantees is that an expert must write the PDDL: encode the domain, identify constraints, and specify the goal in a form a solver can consume. This is a long-standing bottleneck (Kambhampati et al., 2024): people communicate planning intent in natural language, while classical planners require something else.

Large language models offer a tempting bridge: read the natural-language specification, write the PDDL, hand it to a planner (Liu et al., 2023). In practice, LLMs alone are unreliable planners and PDDL authors: they hallucinate predicates, mis-encode constraints, mismatch types, omit initial-state facts, and produce plans that fail validation (Valmeekam et al., 2023; Stechly et al., 2024; Zuo et al., 2024). A growing body of work shows the two are most effective combined: the LLM drafts a formal representation, an external solver and validator check it, and feedback drives iterative repair (Liu et al., 2023; Kambhampati et al., 2024; Guan et al., 2023). What remains open is how to organise that repair.

Recent agentic frameworks (La Malfa et al., 2026) decompose PDDL repair into a pool of specialised agents (syntax fixes, type-hierarchy repair, hallucination stripping, temporal consistency, and so on) coordinated in a closed loop: at each step an orchestrator inspects the current PDDL, solver logs, and validator errors, picks an agent, runs it, and re-tries the planner. This lifts smaller and non-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

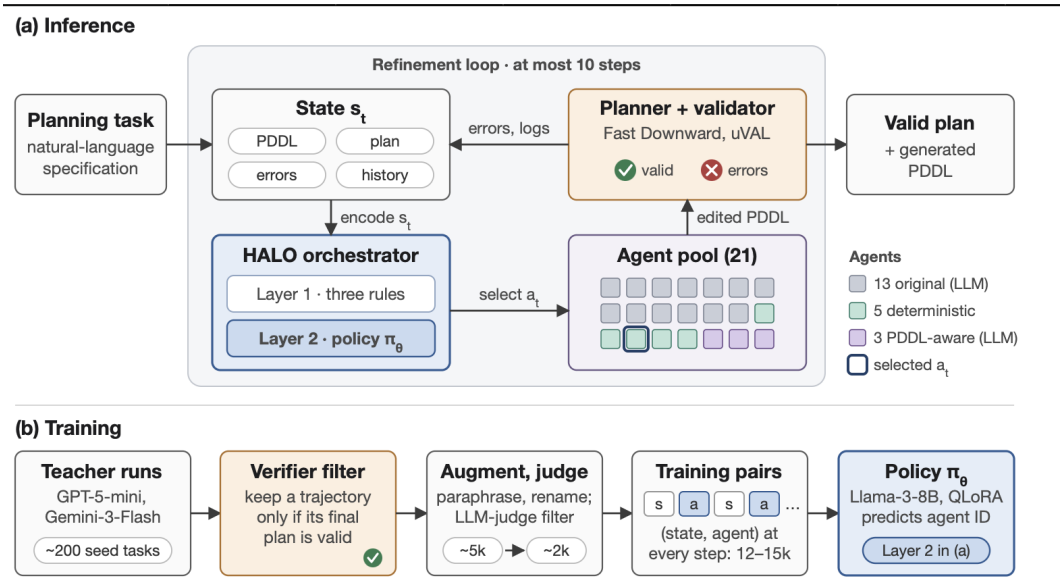


Figure 1: HALO overview. **(a)** Inference: the PDDL pair is refined step by step; at each step Layer 1 rules or the learned policy π_θ (Layer 2) pick one of 21 agents, then the planner and validator rerun. **(b)** Training: teacher trajectories are kept only if the validator accepts their final plan, then augmented and judge-filtered; π_θ is trained to predict the teacher's agent ID at every step.

frontier models to near-frontier accuracy on long-horizon benchmarks (La Malfa et al., 2026). The remaining bottleneck is the orchestrator itself: a prompted frontier LLM (GPT-5, Gemini-3, Claude Sonnet 4.6, and similar) called once per refinement step, whose prompt carries the full PDDL state, so every step pays a long-context frontier-LLM call on top of the repair agent's own work. Can the orchestrator be replaced by a small, locally served model? At first glance this looks hard: there is no ground-truth label for "the correct next agent given the current state," and the decision is several causal steps removed from the only externally verifiable signal, whether the final plan succeeds.

In this paper we train the orchestrator. HALO (Hybrid Agent-Learned Orchestrator) is a small QLoRA-tuned policy paired with a thin hardcoded-rule layer that takes the orchestrator's place in the loop, running locally on a single GPU. The training signal is already present in the framework: agentic PDDL planning is organised around an external planner (Fast Downward, POPF) and validator (VAL, uVAL) that check whether a candidate plan reaches the goal. We keep only teacher trajectories whose final plan the validator accepts and train the policy to reproduce the teacher's agent choice at each step. This is outcome-filtered behavioural cloning: acceptance certifies where a trajectory ended, not that every step in it was necessary. We also extend the framework along two axes: the action space grows to 21 agents with five deterministic plan-repair routines (Armony et al., 2025) and three PDDL-aware LLM agents (Gestrin et al., 2024); and three trivially rule-decidable selections (cold start, syntax errors, valid plan reached) are handled by hardcoded rules before the policy is consulted.

Our contributions are as follows.

- **A trained orchestrator for agentic PDDL planning.** With the agent pool and repair-agent LLM held fixed, a compact learned policy behind three hardcoded rules, in place of the prompted orchestrator, matches the GPT-5-mini-orchestrated framework on two of three benchmark families and exceeds it on the third, at a small fraction of the orchestration cost.
- **Verifier-filtered trajectory supervision.** We train the orchestrator by imitation on teacher trajectories that an external planner and validator accept, a filter that never enters the loss; we show that the filter is essential and that the policy transfers, with bounded loss, to held-out obfuscated-vocabulary domains.

- **An expanded action space with a hybrid policy.** The agent set grows to 21 with five deterministic plan-repair routines and three PDDL-aware LLM agents (Table 1), and three rule-decidable selections are made by hardcoded rules, without which the learned policy alone is far weaker.

2 Background and Related Work

Our work sits at the intersection of agentic frameworks for LLM-driven PDDL planning, imitation learning from solver-validated demonstrations, and trajectory supervision and tool orchestration for LLM agents.

2.1 Agentic end-to-end LLM planning

A growing body of work uses LLMs as front-ends to symbolic planners, translating natural-language descriptions into PDDL and refining the result against an external solver. LLM+P (Liu et al., 2023) establishes the template: an LLM writes the problem file, a classical planner solves, and a small post-processing step returns the plan in natural language. LLM-Modulo (Kambhampati et al., 2024) formalises the broader idea that LLMs cannot plan reliably on their own and should be paired with formal verifiers. NL2Plan (Gestrin et al., 2024) decomposes the full natural-language-to-PDDL pipeline into six sequential steps with per-step LLM calls and validators. Related efforts construct PDDL world models from text, write generalised policies, build authoring benchmarks, and extend these pipelines to numeric fluents, plan repair, and heuristic-guided search (Guan et al., 2023; Silver et al., 2024; Zhang et al., 2024; Stein et al., 2023; Corrêa et al., 2025; Hao et al., 2025); Mahdavi et al. (2024) refine PDDL domains and problems iteratively from environment feedback. Our novelty is not feedback-driven refinement itself but *learning* which repair agent to apply at each step. The most extensive such framework is La Malfa et al. (2026): specialised repair agents (syntax fixer, type fixer, predicate stripper, and so on) plus one dynamic agent, coordinated by a prompted frontier LLM that picks the agent at every step; it lifts smaller models to near-frontier accuracy across PlanBench (Valmeekam et al., 2023), Google Natural Plan (Zheng et al., 2024), and classical benchmarks. Its orchestrator is a frontier-LLM call at every refinement step, so orchestration cost scales with iterations; we retain the structure of La Malfa et al. (2026) but train the orchestrator at its centre, replacing the per-step call with a single-token decision from a small local model.

2.2 Learning planning policies from solver-validated demonstrations

A separate tradition learns planning policies directly from demonstrations that an external solver has validated; behavioural cloning (Pomerleau, 1991) and DAgger (Ross et al., 2011) are its textbook ancestors. Recent instances rank ground actions with graph neural networks trained on planner-generated data, the labels inheriting the planner's correctness guarantees (Mangannavar et al., 2025b; Mangannavar, 2026). Verifier-guided distillation has also reached LLM reasoning and code generation (Chen et al., 2024a; Kumar et al., 2024; Ahmad et al., 2025). These methods select the next *plan action* with a custom architecture over a domain-specific state representation; we instead select the next *agent* in a refinement loop with a generic language model over a tokenised state, with verifier acceptance used only to filter training trajectories.

2.3 Trajectory supervision and tool orchestration for LLM agents

Fine-tuning LLMs to act as agents has produced a steady stream of methods such as AgentTuning (Zeng et al., 2023), FireAct (Chen et al., 2023), and their successors (Chen et al., 2024b; Song et al., 2024; Zhu et al., 2024; Qi et al., 2024). The pattern is consistent: a strong teacher acts, the student imitates, and the supervision is the full text of what the teacher produced. By contrast, our action space is closed and discrete (21 agent IDs), our supervision is a single token per decision, and trajectories are kept only when an external planner and validator accept the final plan, a filter that open-ended agent pipelines lack. A related line lets an LLM orchestrate a fixed tool library over multiple steps (Yao et al., 2023; Shen et al., 2023; Zhuang et al., 2024; Erdogan et al., 2024), or search over its own actions as in LATS (Zhou et al., 2023). Those settings lack a formal verifier (success is judged by user satisfaction, downstream accuracy, LLM-as-a-Judge scores, or the model's

own value estimates), so they are not like-for-like baselines here; ours is a sound plan validator over a closed repair toolkit.

3 PRELIMINARIES: THE AGENTIC PDDL FRAMEWORK

This section recaps the framework of La Malfa et al. (2026) and fixes notation.

3.1 PLANNING AS ITERATED REFINEMENT

A planning task is specified by a natural-language description σ. A PDDL pair $(\mathcal{D}, \mathcal{P})$ consists of a domain $\mathcal{D}$ (typed objects, predicates, action schemas with preconditions and effects) and a problem $\mathcal{P}$ (initial state, goal). Given $(\mathcal{D}, \mathcal{P})$, a classical planner solves for a plan ρ, a sequence of grounded actions, and a validator V checks whether ρ achieves the goal from the initial state under the action semantics of $\mathcal{D}$. We write $V(\mathcal{D}, \mathcal{P}, \rho) = \text{valid}$ when the plan succeeds and $V(\mathcal{D}, \mathcal{P}, \rho) = e$, an error report, otherwise. Throughout, the planner is Fast Downward (Helmert, 2006) (`lama-first`, 120 s limit), whose plans are checked with the universal validator uVAL; for temporal extensions we use POPF (Coles et al., 2010) with VAL (Howey et al., 2004).

The end-to-end task is to produce $(\mathcal{D}, \mathcal{P}, \rho)$ with $V(\mathcal{D}, \mathcal{P}, \rho) = \text{valid}$ given only σ. A direct LLM-only mapping $\sigma \mapsto (\mathcal{D}, \mathcal{P}, \rho)$ is unreliable (Valmeekam et al., 2023; Stechly et al., 2024). The agentic framework treats this end-to-end task as iterative refinement. At iteration $t \in \{0, 1, \dots\}$ the system holds a state

$$s_t = (\mathcal{D}_t, \mathcal{P}_t, \rho_t, e_t, h_t), \tag{1}$$

where e_t collects validator and planner errors observed so far and h_t is the recent history of agents applied. The framework iterates

$$s_{t+1} \;=\; \text{Apply}\big(s_t,\; \pi(s_t)\big), \tag{2}$$

where π is the orchestrator policy that selects an agent $a \in \mathcal{A}$, and Apply runs the chosen agent on s_t, re-invokes the planner, and re-runs the validator to produce s_{t+1}. The loop ends at the first step whose plan is valid, when the orchestrator selects the termination agent `NoOpAgent`, or at the budget $t = T_{\max}$. In La Malfa et al. (2026) (as implemented in their released code), $\mathcal{A}$ holds 13 agents: 11 LLM-prompted repair routines, each targeting a narrow class of PDDL or plan error, the termination agent, and a natural-language rendering agent (`AgentNaturalLanguage`); Appendix A.1 lists them all.

3.2 THE ORCHESTRATOR DECISION AS A POLICY

In La Malfa et al. (2026), π is a prompted frontier LLM that reads s_t and emits an agent name. We treat π as a *learnable* policy. A trajectory $\tau = (s_0, a_0, \dots, s_T, \rho_T)$ is *valid* if $V(\mathcal{D}_T, \mathcal{P}_T, \rho_T) = \text{valid}$. Each pair (s_t, a_t) of a valid trajectory is a demonstration of a choice that was followed by success, not a certificate that a_t was necessary or optimal; valid trajectories may contain redundant steps.

4 METHOD

This section describes how we replace the prompted orchestrator at the centre of the refinement loop (Section 3.2) with a learnable policy; Figure 1(b) summarises the training pipeline.

4.1 EXPANDED 21-AGENT ACTION SPACE

We extend the agent pool $\mathcal{A}$ from the 13 agents of La Malfa et al. (2026) (as implemented in their released code) to $|\mathcal{A}| = 21$ by adding five deterministic plan-repair routines and three PDDL-aware LLM agents (Table 1). Each agent reads the relevant slice of s_t and returns a candidate update, after which the framework reruns the planner and validator. Agent IDs are integers 0–20, each a single token under the Llama-3 tokeniser (verified at startup).

Table 1: Agents added to the pool (IDs 13–20). Left: deterministic plan-repair routines adapted from Armony et al. (2025) (no LLM cost). Right: PDDL-aware LLM agents adapted from Gestrin et al. (2024). The 13 baseline agents (IDs 0–12) and all prompts are in Appendix A.1.

Deterministic plan repair			*PDDL-aware LLM agents*		
13	`VariableSwapper`	Swapped args	18	`TypeHierarchyFixer`	Fix `:types`
14	`ActionReorderer`	Reorder by deps.	19	`PredicateGeneralizer`	Generalise types
15	`RedundancyStripper`	Drop dup./no-op	20	`InitialStateSuggester`	Add `:init` facts
16	`PlanTruncator`	Truncate at goal			
17	`SubplanFiller`	Fill gaps (planner)			

Baseline agents (IDs 0–12). Thirteen baseline agents from La Malfa et al. (2026); most are LLM-prompted repair routines, with `NoOpAgent` serving as a (non-LLM) termination signal and `AgentNaturalLanguage` rendering the final plan (full list in Appendix A.1).

Deterministic plan-repair agents (IDs 13–17). Five non-LLM routines adapted from Armony et al. (2025), operating directly on a candidate plan when the planner returns one but validation fails (Table 1). They run in milliseconds and contribute zero LLM cost.

PDDL-aware LLM agents (IDs 18–20). Three LLM-prompted agents adapted from Gestrin et al. (2024), each addressing a PDDL-modelling failure that the baseline 13 do not cover: type-hierarchy repair, predicate-type generalisation, and initial-state completion (Table 1).

4.2 Verifier-filtered trajectory collection

We construct training data by running a strong prompted teacher orchestrator (GPT-5-mini for the headline pipeline; we also collect with Gemini-3-Flash for diversity) over training problems from 11 PDDL domains. Each rollout produces a trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$ that terminates either when $V(\mathcal{D}_t, \mathcal{P}_t, \rho_t) = \text{valid}$, when the teacher selects `NoOpAgent`, or at the iteration budget $T_{\max} = 10$. As in La Malfa et al. (2026), the teacher makes every choice itself. At every step we log s_t, the selected agent a_t, the agent's response, the post-execution state s_{t+1}, and the validator metrics before and after.

We then filter the trajectory pool in three passes.

(1) Verifier filter (hard). A trajectory survives only if $V(\mathcal{D}_T, \mathcal{P}_T, \rho_T) = \text{valid}$ for its terminal state. Partial trajectories that exhaust the iteration budget without producing a valid plan are discarded. This is the only step in the pipeline that touches the verifier directly; it acts as a binary data-acceptance gate, not as a loss signal.

(2) Specification-level augmentation. Surviving trajectories are augmented by paraphrasing the natural-language specification, renaming objects, and varying goal-formulation surface forms, yielding $\approx$ 5k trajectories from $\approx$ 200 verifier-accepted seed problems across the 11 domains. Seed problems are excluded from evaluation.

(3) LLM-as-judge filter (soft). A frontier LLM rates each augmented trajectory on per-step coherence and final-plan quality, and trajectories below a fixed score threshold are dropped, leaving $\approx$ 2k. This partly addresses the fact that the hard filter certifies only the outcome: it removes accepted trajectories whose intermediate steps are erratic or redundant.

The supervised set is every (s_t, a_t) pair from the surviving trajectories, including steps that a Layer-1 rule would resolve at inference (Section 4.3), giving roughly 12–15k pairs.

4.3 Hybrid hardcoded + learned policy

At inference time the orchestrator runs in two layers.

Layer 1: hardcoded rules. Three orchestration decisions are trivially derivable from s_t alone, and we resolve them by rule without consulting any model: an empty domain (cold start) selects `AgentEmergency`, a syntax error in e_t selects `AgentSyntaxPDDL`, and a valid plan selects `NoOpAgent` (terminate). Algorithm 1 gives the full inference loop.

Algorithm 1 HALO inference loop ($T_{\max} = 10$; s_0 has empty PDDL, plan, errors, and history).

for $t = 0, 1, \ldots, T_{\max} - 1$ **do**
  if $\mathcal{D}_t = \emptyset$ **then** $a_t \leftarrow$ `AgentEmergency` ▷ Layer 1: cold start
  else if e_t contains a syntax error **then** $a_t \leftarrow$ `AgentSyntaxPDDL` ▷ Layer 1: syntax
  else $a_t \leftarrow \arg\max_{i \in \{0,\ldots,20\}} \pi_\theta\big(i \mid \mathrm{encode}(s_t)\big)$ ▷ Layer 2: learned policy
  $(\mathcal{D}_{t+1}, \mathcal{P}_{t+1}) \leftarrow a_t(s_t)$; $\rho_{t+1} \leftarrow \mathrm{Planner}(\mathcal{D}_{t+1}, \mathcal{P}_{t+1})$; $e_{t+1} \leftarrow V(\mathcal{D}_{t+1}, \mathcal{P}_{t+1}, \rho_{t+1})$
  if $e_{t+1} = \mathrm{valid}$ **then return** $(\mathcal{D}_{t+1}, \mathcal{P}_{t+1}, \rho_{t+1})$ ▷ Layer 1: `NoOpAgent`
  $s_{t+1} \leftarrow (\mathcal{D}_{t+1}, \mathcal{P}_{t+1}, \rho_{t+1}, e_{t+1}, h_t \| a_t)$
return failure ▷ budget exhausted

Layer 2: learned policy. If no rule fires, the trained policy π_θ runs on the encoded state and produces a single token from $\{0, \ldots, 20\}$ by greedy decoding, mapped to the corresponding agent ID.

Layer 1 handles these decisions at zero cost. The policy is trained on these states too, so running it without Layer 1 (Table 4) measures the learned policy alone, with no train–test mismatch. The prompted baselines follow La Malfa et al. (2026) and have no Layer 1: the prompted LLM makes every decision, including termination.

4.4 Supervised fine-tuning

Input encoding. The state s_t is rendered into a structured prompt with eight sections: the natural-language specification, the current PDDL domain, the current PDDL problem, the current plan, validator errors, planner logs, the recent history h_t (up to ten prior agents), and the available-agent list (Appendix A.3). Long sections are cut to a character cap, and a prompt that still exceeds 2048 tokens after the Llama-3 chat template is applied is truncated from the left.

Output and loss. The target is a single agent-ID token $a^\star \in \{0, \ldots, 20\}$. All prompt tokens carry $\mathrm{label} = -100$ so they do not contribute to the loss; only the agent-ID token does:

$$\mathcal{L}(\theta;\, s, a^\star) \;=\; -\log \pi_\theta\big(a^\star \mid \mathrm{encode}(s)\big). \tag{3}$$

Architecture and optimisation. The base model is fine-tuned with QLoRA (Dettmers et al., 2023) ($r = 16$, $\alpha = 32$, dropout 0.05) on the attention and MLP projections, with the LM head also trained, AdamW, an effective batch of 16, and three epochs; one Llama-3-8B run takes ≈ 6 wall-clock hours on an $8\times$ A100 node. Full hyperparameters are listed in Appendix A.2.

Inference. The trained policy runs with greedy decoding at temperature 0 on a single 24 GB GPU.

5 Experimental Setup

We ask whether HALO matches prompted orchestrators at lower cost (**Q1**), transfers to held-out domains (**Q2**), needs the verifier filter (**Q3**), and needs the rules and the added agents (**Q4**).

5.1 Benchmarks

We follow the benchmark set of La Malfa et al. (2026), spanning 11 distinct PDDL domains across three benchmark families: (1) *PlanBench* (Valmeekam et al., 2023): blocksworld, depots, logistics, mystery blocksworld, and obfuscated deceptive logistics; (2) *Google Natural Plan* (Zheng et al., 2024): calendar scheduling, meeting planning, and trip planning; (3) *Classical planning*: Blocksworld, Hanoi, and the Borealis suite (childsnack, floortile) for IPC-style coverage.[1] The Blocksworld PDDL domain appears in both the PlanBench and classical suites; the 11-domain count counts it once. In every family the input is the natural-language task and the system writes its own

[1] Problem counts. PlanBench (2,270): blocksworld 500, depots 500, logistics 285, mystery blocksworld 500, obfuscated deceptive logistics 285, plus 100-problem `blocksworld_3` and `mystery_blocksworld_3` subsets. Natural Plan (3,600): calendar 1,000, meeting 1,000, trip 1,600. Classical (248): Blocksworld easy/medium/hard and Hanoi easy/medium/hard/extreme (30 each), childsnack 19, floortile 19.

PDDL, including for Natural Plan, which has no reference PDDL; success therefore means that the validator accepts this formalisation (Section 5.3). We evaluate on every problem not used as a training seed.

5.2 Baselines and trained models

Baselines. Prompted baselines run the agentic framework of La Malfa et al. (2026) with GPT-5-mini or Gemini-3-Flash as the orchestrator, which makes every decision, as in the original framework. Every configuration, HALO included, shares the same 21-agent pool, the same LLM for all LLM-backed repair agents (Gemini-2.5-Flash), the same planner and validator, and $T_{\max} = 10$. The configurations therefore differ in the orchestrator: a prompted LLM, or HALO's rules plus learned policy. Restricting the pool to the original 13 agents is an ablation (Table 4).

HALO. The policy π_θ is Llama-3-8B-Instruct, fine-tuned as in Section 4.4.

5.3 Metrics

Success (validator acceptance). A problem counts as solved when the validator accepts the final *generated* PDDL pair and plan. This measures plan validity under the system's own formalisation, not fidelity of that formalisation to the natural-language task: a domain or problem that an agent has weakened (a dropped constraint, an added `:init` fact) can still yield an accepted plan. We do not check generated PDDL against reference PDDL or gold answers, so absolute success rates are an upper bound on task accuracy and are not comparable to the judge-against-gold numbers of La Malfa et al. (2026). Because all configurations share the agents, agent LLM, planner, and validator, the *relative* comparisons in Tables 2–4 are like for like, with two caveats discussed in Section 7. A family's score is the unweighted mean of its per-domain success rates.

Efficiency. We report mean refinement iterations; LLM calls per episode, counting calls to the prompted orchestrator and to LLM-backed agents (HALO's local forward passes are not API calls); and the provider-billed tokens of the orchestrator's own calls for the prompted baselines (April–May 2026 rates), and energy plus amortised hardware for HALO's local forward passes. Agent LLM calls are excluded for every configuration; since all configurations use the same agent LLM, that cost differs between them only through the number of agent calls, which the calls column reports.

5.4 Compute and statistics

Training uses one $8\times$A100 80 GB node ($\approx$ 6 hours per run); HALO inference uses a single 24 GB GPU. Every number is a single run per configuration. HALO decodes greedily and is deterministic; the prompted baselines are not, and we could not afford repeated API runs. For scale, binomial 95% intervals on a single run are about ± 1 point on PlanBench and Natural Plan (thousands of problems) and about ± 3 points on Classical (248); they cover the sampling of problems, not the run-to-run variance of the stochastic baselines, so we treat differences under about 3 points as ties.

6 Results and Discussion

6.1 Main results: HALO vs prompted baselines

Table 2 reports results aggregated per benchmark family (Q1).

Success rate. Against the GPT-5-mini-orchestrated system, HALO is level on PlanBench and Classical (+3.3 and +2.8 points, within the tie margin of Section 5.4) and far ahead on Natural Plan (+37.3), where GPT-5-mini reaches only 51.5%. Against Gemini-3-Flash, HALO is 2.9 points lower on both PlanBench and Natural Plan; its Classical figure covers only the Borealis subset, so we draw no conclusion from that column. Every row shares the agents and agent LLM, so these differences come from the orchestrator; since HALO's orchestrator includes the Layer-1 rules and the prompted ones do not, part of the gap may be due to the rules (Section 6.3).

Iterations and LLM calls. HALO needs the fewest iterations and LLM calls per episode in every family: 46–54% fewer calls than GPT-5-mini and about 40% fewer than Gemini-3-Flash. Both baselines make about 1.7 LLM calls per refinement step, against 1.05–1.14 for HALO, which also

Table 2: Main results by benchmark family (single run per configuration). All rows share the 21-agent pool and the agent LLM; the prompted rows are the framework of La Malfa et al. (2026), HALO is Layer-1 rules plus the learned policy. Succ: validator success, the mean per-domain % of problems whose final generated PDDL and plan the validator accepts (fidelity to the NL task is not checked; §5.3). Iter: mean refinement steps. Calls: LLM calls per episode. Cost: orchestration cost per task in US cents (HALO: amortised local GPU). Higher is better for Succ, lower for the rest. *Borealis subset only (childsnack, floortile), not comparable to the full-family figures.

	PlanBench				Natural Plan				Classical			
Orchestrator	Succ	Iter	Calls	Cost	Succ	Iter	Calls	Cost	Succ	Iter	Calls	Cost
GPT-5-mini	91.6	2.8	4.8	18.0	51.5	4.0	6.8	22.0	93.0	2.8	4.8	18.0
Gemini-3-Flash	**97.8**	2.2	3.7	6.0	**91.7**	3.2	5.4	8.0	(93.0)*	2.5	4.3	6.0
HALO (ours)	94.9	**2.1**	**2.2**	**0.4**	88.8	**2.8**	**3.2**	**0.4**	**95.8**	**2.3**	**2.6**	**0.4**

Table 3: Held-out transfer (success, %). Policy retrained without the two obfuscated-vocabulary domains, evaluated on them zero-shot. In-dist. is the full model.

Domain	In-dist.	Held-out	Δ
Mystery BW	91	84	−7
Obf. dec. logist.	87	76	−11

Table 4: Ablations (success, %). −verifier filter and −8 agents retrain the policy without that component; −rules runs the full policy (trained on all steps) without Layer 1, i.e. the learned policy alone.

Configuration	PlanB.	NatPlan	Class.
HALO (full)	94.9	88.8	95.8
− verifier filter	39.6	53.1	47.8
− Layer-1 rules	61.3	48.3	49.9
− 8 added agents	67.3	41.2	58.3

takes fewer steps. A prompted orchestrator pays one API call per step on top of the agent's own call, whereas HALO's decision is a local forward pass or a rule.

Cost. Orchestration cost falls from 18–22 cents per task for GPT-5-mini and 6–8 cents for Gemini-3-Flash to 0.4 cents for HALO, roughly 45× and 15× lower. This covers orchestration only; agent LLM calls, which every configuration makes, are not included. Training HALO is a one-off cost of about 48 A100-hours (Appendix A.5).

6.2 Cross-domain generalisation

To test whether the policy relies on domain vocabulary (Q2), we hold out the two obfuscated-vocabulary domains (mystery blocksworld and obfuscated deceptive logistics), retrain the policy on the remaining nine domains, and evaluate it on the held-out pair zero-shot (Table 3). Each held-out domain shares its planning structure with an in-distribution sibling (blocksworld, logistics) but replaces the entire surface vocabulary, so any success must come from structural features (validator errors, solver logs, refinement history) rather than lexical overlap; this is also the condition under which La Malfa et al. (2026) report the largest degradation for the prompted pipeline.

Withholding a domain costs 7 points on mystery blocksworld (91 to 84) and 11 on obfuscated deceptive logistics (87 to 76): the policy keeps most of its in-distribution success when the vocabulary is replaced, and the actively misleading vocabulary shows the larger drop. This tests vocabulary transfer with a structural sibling in training, not transfer to structurally new domains, which we leave open (Section 7).

6.3 Ablations

Table 4 isolates the three design decisions (**Q3**–**Q4**). **Verifier filtering is essential.** Training on unfiltered teacher trajectories collapses success (94.9 to 39.6 on PlanBench, with comparable drops elsewhere): the model imitates choices from failed episodes as readily as from successful ones. **Rules.** The policy is trained on every step, including those the rules resolve, so running it without Layer 1 measures the learned policy alone. Success drops by 34–46 points, below both prompted orchestrators, which make every decision themselves. The learned policy on its own is therefore

not a replacement for a prompted orchestrator; HALO's results are a property of the hybrid, and the three rules, which cost nothing to run, are an essential part of the design. We did not run the prompted orchestrators behind the same rules, so we cannot say how much the rules alone would improve them. **21 vs. 13 agents.** A policy trained and run with only the 13 original agents loses ground in every family, most on Natural Plan (88.8 to 41.2) and Classical (95.8 to 58.3).

6.4 Discussion

Why HALO can beat GPT-5-mini. HALO outscores the GPT-5-mini-orchestrated framework although its policy imitates teacher choices. Three factors plausibly contribute. The verifier filter keeps only the teacher's successful episodes, so the student imitates the teacher's best behaviour rather than its average; the Layer-1 rules make three decisions that the prompted orchestrator must get right on its own; and the training pool also contains Gemini-3-Flash trajectories (Section 4.2), the far stronger teacher on Natural Plan. HALO is thus best read as distilling the better of its two teachers into a local model at a fraction of the cost; it does not surpass Gemini-3-Flash where both are measured on the full family.

7 Limitations and Future Work

Validity is not fidelity. Our success metric is validator acceptance of the *generated* PDDL and plan (Section 5.3). Several agents edit the domain and problem, adding `:init` facts, stripping unsupported features, or merging objects (IDs 7, 8, 20; Appendix A.1), so a plan can be accepted for a formalisation weaker than the task. Shared agents make the comparisons like for like, with one caveat: a learned policy could favour such agents more than a prompted orchestrator does, and we have not measured this. Checking generated PDDL against reference PDDL, as Planetarium (Zuo et al., 2024) and plan-set comparison (Oswald et al., 2024) do, or final plans against Natural Plan's gold answers, is the most important next evaluation.

No prompted-plus-rules baseline. The prompted baselines follow La Malfa et al. (2026) and run without Layer 1, while the learned policy alone is far weaker than they are (Table 4). Our comparison is therefore between complete orchestrators; a prompted orchestrator behind the same rules would separate the contribution of the rules from that of the learned policy.

Single runs. Every configuration is run once; margins under about 3 points should be read as ties.

The teacher bounds the strategy. HALO can only express agent-selection patterns that appear in accepted teacher trajectories; new strategies need a signal beyond imitation.

Transfer is to new vocabulary, not new structure. Both held-out domains have a structural sibling in training; withholding them costs 7 and 11 points, and structurally unseen IPC domains are untested.

The planner is a ceiling. When Fast Downward times out we cannot tell an unsolvable formalisation from an insufficient search budget, so such trajectories are discarded, as in La Malfa et al. (2026).

Future work. RL with validator acceptance as the reward could move past the teacher, and the cheap policy makes search over orchestration choices, pruned by the validator, affordable, as learned policies have enabled elsewhere (Mangannavar et al., 2025a; Mangannavar & Tadepalli, 2026).

8 Conclusion

HALO replaces the prompted frontier-LLM orchestrator of agentic PDDL planning frameworks with a small QLoRA-tuned policy behind three hardcoded rules, trained by imitating teacher trajectories that the validator accepts. With agents and agent LLM held fixed, HALO matches the GPT-5-mini-orchestrated framework on PlanBench and classical planning, exceeds it on Natural Plan, comes within three points of Gemini-3-Flash, and cuts orchestration cost roughly 45× and 15×. Two components are essential: the verifier filter, without which success falls to 40–53%, and the rules, without which the learned policy falls below the prompted orchestrators.

REFERENCES

Wasi Uddin Ahmad et al. Opencodereasoning: Advancing data distillation for competitive coding. *arXiv preprint*, 2025.

Ma'ayan Armony, Albert Meroño-Peñuela, and Gerard Canal. How far are llms from symbolic planners? an nlp-based perspective, 2025. URL https://arxiv.org/abs/2508.01300.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

Guoxin Chen et al. Alphamath almost zero: Process supervision without process. *arXiv preprint arXiv:2405.03553*, 2024a.

Zehui Chen et al. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*, 2024b.

Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2010.

Augusto B Corrêa, André G Pereira, and Jendrik Seipp. Classical planning with llm-generated heuristics: Challenging the state of the art with python code. *arXiv preprint arXiv:2503.18809*, 2025.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023. URL https://arxiv.org/abs/2305.14314.

Lutfi Eren Erdogan, Nicholas Lee, Siddharth Jha, Sehoon Kim, Ryan Tabrizi, Suhong Moon, Coleman Richard Charles Hooper, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Tinyagent: Function calling at the edge. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 80–88, 2024.

Elliot Gestrin, Marco Kuhlmann, and Jendrik Seipp. Nl2plan: Robust llm-driven planning from minimal text descriptions. *arXiv preprint arXiv:2405.04215*, 2024.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. *NeurIPS*, 2023.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. Large language models can solve real-world planning rigorously with formal verification tools, 2025. URL https://arxiv.org/abs/2404.11891.

Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246, 2006.

Richard Howey, Derek Long, and Maria Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 294–301, 2004.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, et al. Llms can't plan, but can help planning in llm-modulo frameworks. *arXiv preprint arXiv:2402.01817*, 2024.

Aviral Kumar et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024.

Emanuele La Malfa, Ping Zhu, Samuele Marro, Sara Bernardini, and Michael Wooldridge. End-to-end pddl planning with hardcoded and dynamic agents. *arXiv preprint arXiv:2512.09629*, 2026. v2, May 2026.

Bo Liu, Yuqian Jiang, Xiaohan Zhang, Qiang Liu, Shiqi Zhang, Joydeep Biswas, and Peter Stone. Llm+p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477*, 2023.

Sadegh Mahdavi, Raquel Aoki, Keyi Tang, and Yanshuai Cao. Leveraging environment interaction for automated pddl translation and planning with large language models. *Advances in Neural Information Processing Systems*, 37:38960–39008, 2024.

Rajesh Mangannavar and Prasad Tadepalli. Gammazero: Learning to guide belief-space search for long-horizon pomdps with generalizable graph representations. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 36, pp. 658–666, 2026.

Rajesh Mangannavar, Alan Fern, and Prasad Tadepalli. Hierarchical object-oriented pomdp planning for object rearrangement, 2025a. URL https://arxiv.org/abs/2412.01348.

Rajesh Mangannavar, Stefan Lee, Alan Fern, and Prasad Tadepalli. Graph neural network based action ranking for planning. In *Advances in Neural Information Processing Systems*, 2025b.

Rajesh D Mangannavar. Scalable planning through abstraction and learning in fully and partially observable domains. 2026.

Drew McDermott et al. Pddl: The planning domain definition language. *Technical Report*, 1998.

James Oswald, Kavitha Srinivas, Harsha Kokel, Junkyu Lee, Michael Katz, and Shirin Sohrabi. Large language models as planning domain generators (student abstract). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 23604–23605, 2024.

Dean A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

Zehan Qi et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. *arXiv preprint arXiv:2411.02337*, 2024.

Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 627–635, 2011.

Yongliang Shen, Kaitao Song, et al. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *NeurIPS*, 2023.

Tom Silver, Soham Dan, Kavitha Srinivas, Josh Tenenbaum, Leslie Kaelbling, and Michael Katz. Generalized planning in pddl domains with pretrained large language models. In *AAAI*, 2024.

Yifan Song et al. Trial and error: Exploration-based trajectory optimization for llm agents. *arXiv preprint arXiv:2403.02502*, 2024.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. Chain of thoughtlessness: An analysis of cot in planning. *arXiv preprint arXiv:2405.04776*, 2024.

Katharina Stein, Daniel Fiser, Jorg Hoffmann, and Alexander Koller. Autoplanbench: Automatically generating benchmarks for llm planners from pddl. *arXiv preprint arXiv:2311.09830*, 2023.

Karthik Valmeekam, Matthew Marquez, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. In *Advances in Neural Information Processing Systems*, volume 36, pp. 38975–38987, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, et al. Tree of thoughts: Deliberate problem solving with large language models. *NeurIPS*, 2023.

Aohan Zeng, Mingdao Liu, Rui Lu, et al. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.

Tianyi Zhang, Li Zhang, et al. Proc2pddl: Open-domain planning representations from texts. *arXiv preprint arXiv:2403.00092*, 2024.

Huaixiu Steven Zheng, Swaroop Mishra, Hugh Zhang, Xinyun Chen, Minmin Chen, Azade Nova, Le Hou, Heng-Tze Cheng, Quoc V. Le, Ed H. Chi, and Denny Zhou. NATURAL PLAN: Benchmarking LLMs on natural language planning, 2024. URL https://arxiv.org/abs/2406.04520.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024. *URL https://arxiv. org/abs/2310.04406*, 4, 2023.

Yuqi Zhu, Shuofei Qiao, et al. Knowagent: Knowledge-augmented planning for llm-based agents. *arXiv preprint arXiv:2403.03101*, 2024.

Yuchen Zhuang, Xiang Chen, et al. Toolchain*: Efficient action space navigation in large language models with a* search. In *ICLR*, 2024.

Max Zuo et al. Planetarium: A rigorous benchmark for translating text to structured planning languages. *NeurIPS Workshop*, 2024.

A Appendix

A.1 Full agent set with prompts

This appendix lists every agent in our 21-agent expanded action space (main paper, Method section) with its full system prompt where applicable. LLM-prompted agents use the verbatim prompts shown below. Deterministic agents (IDs 13–17) are non-LLM routines and are described by what they do rather than a prompt.

A.1.1 Baseline agents (IDs 0–12)

These thirteen agents are inherited from La Malfa et al. (2026) (as implemented in their released code). Most are invoked as an LLM call with the system prompt below and the appropriate slice of s_t as the user prompt; `NoOpAgent` (ID 11) is a non-LLM termination signal.

Agent 0: `AgentHallucinations`.

```
You are a meticulous multi-agent planning engineer. You specialize in
    eliminating hallucinations from PDDL domains and problems by
    checking every predicate, object, and goal against the ground
    specification. Preserve valid constructs, repair or remove
    unsupported ones, and keep the syntax compliant with classical
    PDDL. Always encode stated busy intervals and preferences (e.g.,
    "avoid", "would rather not", "earliest") as hard constraints in the
    resulting PDDL. Limit the :requirements list to those supported by
    Fast Downward (:typing, :negative-preconditions); never introduce
    :fluents, axioms, conditional effects, or durative constructs.
    Remove placeholder artefacts such as '...', 'None', or 'TBD', and
    ensure that any cost modelling uses (increase ...) effects instead
    of ':cost' headers. If :action-costs remains in the requirements
    list, verify that matching increase effects exist; otherwise drop
    the requirement.
```

Agent 1: `AgentDeepThinkPDDL`.

```
You are an expert Planning Domain Definition Language (PDDL)
    programmer. You analyze the provided domain, problem, and plan
    against the human specification, identify every inconsistency, and
    return corrected PDDL artifacts that satisfy the goal. Treat all
    stated busy intervals and temporal preferences as hard constraints
    and ensure only feasible time windows remain selectable. Verify
    that time durations exactly match the specification, that the total
    time horizon is preserved, and that only direct connections
```

```
    explicitly listed are available. Do not introduce unsupported Fast
    Downward features (:fluents, axioms, conditional effects, durative
    actions, etc.). Remove placeholder tokens such as '...' or 'None',
    limit :requirements to (:strips :typing :negative-preconditions)
    plus :action-costs only when matching (increase ...) effects are
    present, and ensure costs are modelled via effects rather than
    ':cost' headers. When solver logs or syntax errors highlight
    problems (invalid requirements, unknown tokens, malformed numeric
    expressions, stray parentheses), treat them as hard failures to fix
    immediately before returning new PDDL.
```

Agent 2: `AgentEmergency`.

```
You are an emergency response agent for PDDL pipelines. The previous
    refinement loop failed to produce any valid plan and the solver
    crashed with parser/validator errors. Your job is to resolve every
    syntax or semantic issue reported in the logs and deliver a clean
    PDDL 2.1 domain and problem that are fully compatible with the
    specified target solver (Fast Downward style classical STRIPS
    features only). Never introduce unsupported requirements
    (:durative-actions, :fluents, axioms, conditional effects, etc.).
    Fix missing predicates, mismatched parentheses, numeric fluents,
    inconsistent names, and malformed metrics. Encode all constraints
    explicitly, ensure types/objects/predicates match across domain and
    problem, and keep action schemas coherent with the human and JSON
    specifications. Treat every warning/error as mandatory.
```

Agent 3: `AgentEmergencySolution`.

```
You are a senior planning analyst tasked with providing a concrete
    natural language solution outline when automated planning fails.
    Your suggestions guide downstream agents in regenerating compliant
    PDDL.
```

Agent 4: `AgentDeepThinkConstraints`.

```
You are an expert Planning Domain Definition Language (PDDL)
    programmer. Analyze the PDDL domain and problem against the
    natural-language and JSON specifications. Focus on whether each
    agent's constraints are correctly captured as PDDL formulae,
    especially agents' availability (e.g., meeting durations or robot
    schedules), and any natural-language preferences that must be
    enforced. Ensure time durations sum to the required actions length,
    highlight irreconcilable demands, and verify that the domain
    declares only Fast Downward compatible requirements (:typing,
    :negative-preconditions, and :action-costs only when justified by
    (increase ...) effects). Remove placeholder tokens such as '...' or
    'None', and rewrite any ':cost' headers as (increase ...) effects.
```

Agent 5: `AgentTemporalConsistency`.

```
You are a planning engineer focused on temporal and causal consistency.
    Ensure that enumerated phases, resource limits, and ordering
    constraints from the specification are encoded as hard
    requirements. Eliminate bookkeeping shortcuts (quota tokens, "pay
    shortfall" actions, optional penalty payments, or oscillating
    transitions) that allow violations after the fact. Model discrete
    time or stage progression explicitly (e.g., via ordered objects and
    successor predicates), require contiguous occupancy when durations
    are specified, and enforce terminal conditions exactly as
    described. Keep :requirements within (:strips :typing
```

```
    :negative-preconditions) unless a matching (increase ...) effect
    justifies :action-costs. Never leave placeholders or assume extra
    connectivity.
```

Agent 6: `AgentEnforceMultiAgency`.

```
You are an expert Planning Domain Definition Language (PDDL)
    programmer. Ensure that the PDDL domain and problem correctly
    represent each agent's actions as distinct entities. Encode every
    stated constraint or preference as a structural restriction so that
    no agent executes an action that violates availability or timing
    requirements. Reject candidate models that invent indirect actions,
    connections or unsupported requirements (:fluents, axioms,
    conditional effects). Confirm that time durations of each action
    and the overall action horizon match the natural-language brief
    before approving the domain/problem pair. Remove placeholder tokens
    (e.g., '...', 'None'), limit :requirements to (:strips :typing
    :negative-preconditions) with :action-costs only when justified by
    (increase ...) effects, and rewrite any ':cost' headers as proper
    increase effects.
```

Agent 7: `AgentFastDownwardsAdapter`.

```
You are an agent that adapts PDDL domains and problems for the Fast
    Downward planner. Convert numeric, temporal, or durative features
    into classical STRIPS/ADL-style constructs so the solver can
    operate on the domain while preserving as much of the original
    semantics as possible. Never relax agents' availability, duration
    requirements, or stated preferences when adapting. Restrict the
    :requirements list to features supported by Fast Downward (:typing,
    :negative-preconditions); strip out :fluents, axioms, conditional
    effects, or durative constructs entirely. Remove placeholder tokens
    such as '...' or 'None', and convert any ':cost' headers into
    explicit (increase ...) effects, keeping :action-costs only when
    such effects remain.
```

Agent 8: `AgentReduceVariables`.

```
You are a classical planning engineer focused on trimming the initial
    state cardinality. Rewrite the PDDL domain/problem so that the
    :init section contains as few predicate instances as possible
    without altering the real constraints described by the
    specification. Strategies include removing unused objects, merging
    symmetric resources, introducing helper types or predicates that
    encode invariants once, and ensuring duplicate or implied facts are
    dropped. Maintain Fast Downward compatibility (:typing,
    :negative-preconditions, optional :action-costs only when justified
    by matching (increase ...) effects) and never relax the
    specification's constraints.
```

Agent 9: `AgentSyntaxPDDL`.

```
You are an agent that evaluates each step of a plan. Analyze the
    provided plan against the human specification and identify every
    inconsistency with the PDDL syntax expected by the
    *{target_solver}* planner. Confirm that availability and preference
    constraints remain encoded correctly after your edits and that the
    :requirements list only contains features supported by Fast
    Downward (:typing, :negative-preconditions, and :action-costs only
    when (increase ...) effects remain). Remove placeholder tokens such
```

```
    as '...' or 'None', and ensure action schemas never rely on ':cost'
    headers---instead express costs with (increase ...) effects.
```

Agent 10: `AgentAsynchronicity`.

```
You are an agent that optimizes plans for asynchronicity. Analyze the
    provided plan against the human specification and introduce a
    timestamp variable that marks when each agent performs an action.
    Use it to schedule independent actions concurrently while keeping
    compatibility with the *{target_solver}* planner. Do not violate
    any availability or preference constraint while reshaping the
    schedule, and ensure you do not introduce unsupported Fast Downward
    features (e.g., :fluents, axioms, conditional effects). You may
    think through the problem in as many steps as needed.
```

Agent 11: `NoOpAgent`. Termination signal, not LLM-prompted. Fires when the validator has accepted the current plan; returns `[Info] NoOpAgent called. No more refinements will be performed.` and the framework exits the refinement loop.

Agent 12: `AgentNaturalLanguage`.

```
You are an agent that translates PDDL artifacts into natural language
    instructions. Turn the JSON specification, PDDL problem, PDDL
    domain, and PDDL plan into a clear sequence of human-readable
    actions. Follow the plan provided within the <plan></plan> tags
    closely. You may think through the conversion in as many steps as
    needed.
```

A.1.2 Deterministic plan-repair agents (IDs 13–17)

These five agents are non-LLM routines adapted from Armony et al. (2025); they execute in milliseconds and contribute no LLM cost.

Agent 13: `AgentVariableSwapper`. Fixes wrong variable bindings in PDDL plans by beam-search over permutations of action arguments. Identifies actions with potentially swapped arguments, enumerates candidate permutations under a beam width, and runs each through the validator until one succeeds.

Agent 14: `AgentActionReorderer`. Reorders actions in a plan using circular shifts and dependency-aware swaps. Analyses precondition/effect dependencies, detects out-of-order action pairs, and applies the minimal swap that produces a valid ordering.

Agent 15: `AgentRedundancyStripper`. Removes redundant, duplicate, or oscillating actions. Identifies actions that undo previous effects, duplicate consecutive actions, and actions that do not contribute to the goal. Returns a leaner plan.

Agent 16: `AgentPlanTruncator`. Simulates the plan from the initial state, checks goal satisfaction at every prefix, and truncates the plan at the earliest goal-satisfying state. Useful when the LLM produced extra actions after the goal was already reached.

Agent 17: `AgentSubplanFiller`. Fills gaps in partial plans by invoking Fast Downward on the sub-problem between two valid plan segments. Falls back to a full Fast Downward call if no valid segment exists. The only deterministic agent that itself invokes the planner.

A.1.3 PDDL-aware LLM agents (IDs 18–20)

These three agents are LLM-prompted and adapted from Gestrin et al. (2024). Each addresses a class of PDDL-modelling error that the baseline 13 do not cover.

Agent 18: `AgentTypeHierarchyFixer`.

```
You are a PDDL expert specializing in type hierarchy design. Your task
    is to evaluate and fix the type hierarchy in a PDDL domain model.

Key principles:
- Each subtype should be a MORE SPECIFIC version of its parent type
- Subtypes are NOT physically contained within parent types
- Everything is ultimately a subtype of 'object'
- Types should only appear once in the hierarchy
- New meta-types can be created for organizational purposes

Use this checklist for each type:
1. Does this type fulfill the description of its parent type?
2. Are there types that should be subtypes of this type but aren't?
3. Are any types defined multiple times?

If issues are found, restructure the hierarchy and update the domain
    accordingly. Ensure Fast Downward compatibility with only :typing,
    :strips, :negative-preconditions.
```

Agent 19: `AgentPredicateGeneralizer`.

```
You are a PDDL expert focused on predicate design and type
    generalization.

Your task is to analyze predicates in the domain and generalize their
    parameter types where appropriate. This helps:
- Actions work with subtypes automatically
- Reduces redundant predicate definitions
- Improves domain flexibility

When generalizing:
- Use parent types instead of specific subtypes where semantically
    correct
- Ensure the predicate's meaning is preserved
- Update any affected action preconditions and effects
- Don't over-generalize (e.g., don't use 'object' for everything)

Maintain Fast Downward compatibility with :typing, :strips,
    :negative-preconditions.
```

Agent 20: `AgentInitialStateSuggester`.

```
You are a PDDL expert specialized in initial state completeness
    analysis.

Your task is to identify and add missing facts in the :init section of
    the problem that prevent the planner from finding a solution. Use
    auxiliary domain analysis:

1. For each action, identify what preconditions must be true
2. Trace back what initial facts would enable those preconditions
3. Check if those facts exist in :init
4. Add any missing but logically necessary facts

Important principles:
- Only add facts that MUST be true based on the specification
- Don't add facts that contradict the goal
- Don't add redundant facts
- Ensure type consistency with objects
- Consider symmetric predicates (e.g., if connected(a,b), maybe
    connected(b,a))
```

```
Focus on common missing patterns:
- Location/position initialization for movable objects
- Connectivity/adjacency facts
- Resource availability flags
- Object state initialization (e.g., arm-empty)

Maintain Fast Downward compatibility.
```

A.2 HYPERPARAMETERS

We list the hyperparameters used in the headline runs reported in the main paper (base model: Llama-3-8B-Instruct).

Base model and adapter configuration.

- Base precision: 4-bit NF4 quantisation.
- Adapter and head precision: BF16.
- LoRA rank r: 16. LoRA alpha α: 32. LoRA dropout: 0.05.
- Target modules: `q_proj`, `k_proj`, `v_proj`, `o_proj`, `gate_proj`, `up_proj`, `down_proj`.
- LM head: trained (not frozen).

Optimisation.

- Optimiser: AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.
- Learning rate: 2×10^{-5}.
- Schedule: linear with 100 warmup steps.
- Weight decay: 0 on LoRA adapters.
- Gradient clipping: 1.0.
- Per-device batch size: 4; gradient accumulation: 4 (effective batch 16).
- Epochs: 3. Max sequence length: 2048 tokens.

Data.

- Training domains: 11 PDDL domains (main paper, Benchmarks section).
- Seed problems: ≈ 200.
- Verifier-accepted augmented trajectories: $\approx 5{,}000$.
- After LLM-as-judge soft filter: $\approx 2{,}000$.
- Final supervised set (all steps of the surviving trajectories): $\approx$ 12–15k (state, agent) pairs.

Inference.

- Decoding: greedy (temperature 0, argmax over the agent ID token).
- Max refinement iterations $T_{\max}$: 10.
- Inference hardware: a single 24 GB GPU.

Randomness and seeds. Dataset shuffling and the train/validation split use a fixed seed (42) throughout; inference is deterministic (greedy decoding at temperature 0), so repeated evaluation of a trained checkpoint yields identical results. The prompted teacher and the LLM judge used in the soft filter are sampled through provider APIs and are not seed-controllable.

Software environment. Training on one $8\times$A100 80 GB node; inference on a single 24 GB GPU; PyTorch 2.4.1. The planner is Fast Downward (`lama-first`, 120 s limit) with the uVAL validator, as in the main paper.

A.3 STATE ENCODER DETAILS

The state encoder renders s_t as plain text with the eight sections below, each under a === header, preceded by the instruction "Given the current planning state, select the best agent to invoke next." Long sections are cut to a character cap, keeping their beginning and appending "…". The Llama-3 chat template is then applied, and a prompt longer than 2048 tokens is truncated from the left.

Sections, in order.

1. *Task description*: the natural-language specification σ. Cap: 2,000 characters.
2. *Current domain* $\mathcal{D}_t$, or "(not yet generated)" on cold start. Cap: 2,000 characters.
3. *Current problem* $\mathcal{P}_t$, or "(not yet generated)". Cap: 2,000 characters.
4. *Current plan* ρ_t, or "No valid plan found yet." Cap: 1,000 characters.
5. *Syntax errors* e_t as reported by the planner and validator, or "None". Cap: 500 characters.
6. *Planner logs* (Fast Downward output), omitted when empty. Cap: 500 characters.
7. *Previous agents* h_t: the names of the last 10 agents applied.
8. *Available agents*: the names of all 21 agents.

The prompt terminates with the literal string `Selected Agent:`, which is the position at which the model emits its single agent-ID token. The loss is masked over all prompt tokens (label $= -100$) and computed only on the agent-ID token.

A.4 EXAMPLE TRAJECTORIES

We present one illustrative refinement trajectory per benchmark family. Each lists the agents selected at each step and whether the selection was made by a Layer-1 rule or by the learned Layer-2 policy.

A.4.1 PLANBENCH (BLOCKSWORLD, EASY)

Specification. Two stacking goals over three blocks.

Trajectory.

1. $t = 0$: `AgentEmergency` (rule, empty domain). Initial PDDL drafted.
2. $t = 1$: `AgentSyntaxPDDL` (rule, syntax error). Validator complaint about `:requirements` resolved.
3. $t = 2$: `AgentHallucinations` (learned). Hallucinated predicate removed.
4. $t = 3$: `NoOpAgent` (rule, V accepts).

Outcome: validator accepts after three refinement steps; one learned-policy call.

A.4.2 NATURAL PLAN (MEETING PLANNING)

Specification. Schedule a meeting between four attendees across three time zones with stated busy windows and preferences.

Trajectory.

1. $t = 0$: `AgentEmergency` (rule).
2. $t = 1$: `AgentDeepThinkConstraints` (learned). Busy-window encoding tightened.
3. $t = 2$: `AgentTemporalConsistency` (learned). Ordering invariants added.
4. $t = 3$: `NoOpAgent` (rule).

Outcome: three refinement steps; two learned-policy calls.

A.4.3 CLASSICAL PLANNING (HANOI, MEDIUM, FIVE DISKS)

Specification. Move five disks from peg A to peg C under standard Tower-of-Hanoi rules.

Trajectory.

1. $t = 0$: `AgentEmergency` (rule).
2. $t = 1$: `AgentFastDownwardsAdapter` (learned). Domain rewritten to remove an unsupported requirement.
3. $t = 2$: `AgentSubplanFiller` (learned). Fast Downward used to fill a partial plan.
4. $t = 3$: `NoOpAgent` (rule).

Outcome: three refinement steps; two learned-policy calls.

A.5 COST CALCULATIONS

Scope. The reported cost is *orchestration* cost only. For the prompted baselines it is the provider-billed input and output tokens of every orchestrator call, at April–May 2026 list prices, summed over a family and divided by the number of problems. For HALO it is the energy and amortised hardware cost of its local forward passes. HALO's Layer-1 rules cost nothing. LLM-backed repair agents are excluded everywhere; they run on the same model (Gemini-2.5-Flash) in all configurations. Deterministic agents (IDs 13–17) make no LLM calls.

Per-task figures. GPT-5-mini: 18–22 cents (highest on Natural Plan, whose prompts are longest); Gemini-3-Flash: 6–8 cents; HALO: 0.4 cents, i.e. roughly 45$\times$ and 15$\times$ lower.

One-off costs. Training takes ≈ 48 A100-hours (8 GPUs $\times$ 6 hours), in addition to the API cost of collecting teacher trajectories and running the LLM judge.